

Normalization: How

Large tables with many columns can be difficult to understand and increase redundant data within the database as well as increase the likelihood of anomalies occurring. The normalization process provides database designers with a systematic method of breaking down large tables into a group of related, smaller sized tables which:

- Are easier to understand.
- Minimize the amount of redundant data stored in the database.
- Minimize the likelihood of anomalies occurring within the data.

The normalization process is based on a set of rules for designing relational databases enables and the database to be in one of five states:

- First normal form (1NF)
- Second normal form (2NF)
- Third normal form (3NF)
- Boyce-Codd normal form
- Fourth normal form (4NF)
- Domain-Key normal form

Each normal form represents a set of rules. If a database is in a particular normal form, e.g. third normal form, then all of its tables comply with the rules for the normal form. First, second and third normal forms are based on E.F. Codd's initial set of rules for normalization. Over time additional normal forms have been developed to address problems, which were not addressed by Codd's initial forms. Boyce-Codd normal form is a revision of Codd's initial third normal form. Fourth normal form is an additional set of rules. Domain-Key normal form is a theoretical ultimate in relational database design proposed by Fagin in 1981. There is no defined method to consistently produce tables in Domain-Key normal form, so it remains a work in progress.

Each normal form enables you to create more tables with smaller number of columns in each table. As the number of tables in the database increases data redundancy and the likelihood of an anomaly occurring decreases. This makes the database easier to maintain. Unfortunately, as the number of tables rise the overhead increases. Performance can suffer, particularly if there are high volumes of rows in the tables. Normalization is a trade off between ease of maintenance and performance. Third normal form strikes the best balance (for most applications) between maintenance and performance requirements. Most relational databases are designed to comply with the rules of third normal form.

Denormalization

The term denormalization is used to describe the process where after you have completed a database design that meets the rules of normalization you adjust the design and introduce violations to a normal form.

This may be done to make the database more efficient or easier to use. We will introduce a final database design with the IQ School database that will contain a solution that is not fully normalized.

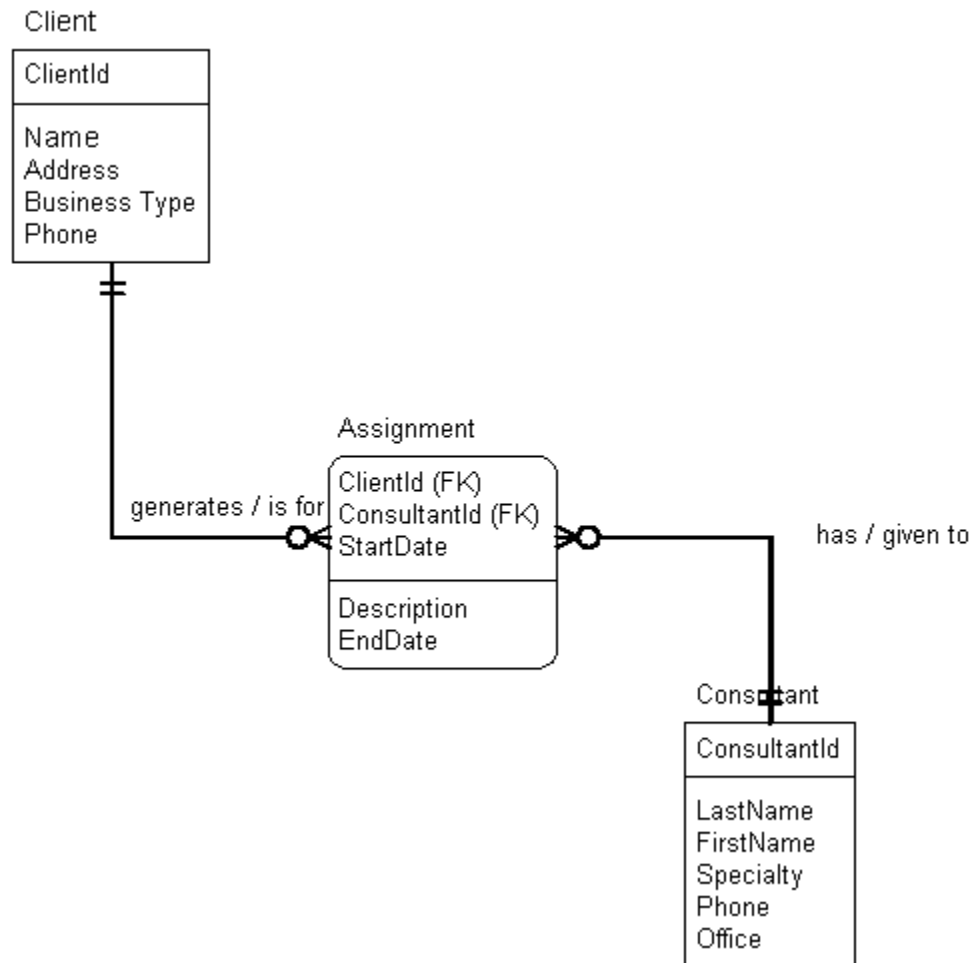
Normal Forms Based on Keys and Business Rules

Before looking at the specific rules of the normal forms we must understand that normal forms are based on two critical concepts.

First, a primary key uniquely identifies a row in a table. Every table has a primary key. If a mistake is made in selecting the primary key for a table, the normalization process will **not** produce a good database design.

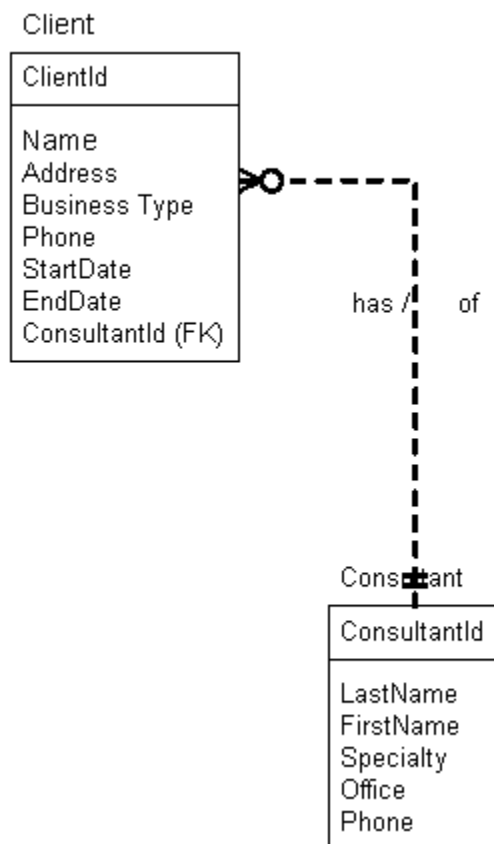
The database design must reflect the business policies (rules) of the organization. The database designer must have a clear understanding of an organization's business rules in order to apply the rules of normalization in a manner that produces a database design, which reflects the business policies of the organization. This is best demonstrated by an example.

Consider the database design for an assignment tracking application for a consulting firm. The database will record information about the firm's consultants, clients and which consultants are assigned to work with a given client. The firm may have a business policy that a client can be served by many different consultants based on a match between the type of service the client requests and the specialty of the consultant. The database design shown below would reflect this business policy.



The business policy results in a many to many relationship between clients and consultants (one consultant can serve many clients and a single client can receive services from many consultants). This causes the normalization process to create the Assignment table with a primary key of ClientId, ConsultantId and StartDate; the combination of ClientId, ConsultantId and StartDate uniquely identifies an individual assignment.

If a different business policy were in place the normalization process would produce a different database design. Suppose the policy was that a client is permanently assigned to one consultant who will provide all services requested by the client. This results in a one-to-many relationship between consultants and clients (a client works with a single consultant, but a single consultant can work with many clients). The following database design would emerge:



Note that the assignment table is no longer required to track which consultant is assigned to which client as there will only be one consultant working with a given client.

The Rules of the Normal Forms

The normalization process lets us systematically break down large tables to many smaller ones. We typically apply the normalization process to each view (subset of data, which facilitates one specific task) in the application. This provides a database design for each individual view. The views are then merged to produce a single logical design for the entire database that accommodates the requirements of all the individual views.

We shall now work through an example view and apply the rules of first, second and third normal forms to produce a schema for the view in third normal form. We start with a source document of some sort, which describes the data required by the view, and use it to define a single table that contains all the data required by the view. We then apply the rules of normalization (first, second then third normal form) to break the single table into many smaller, related tables.

Example: Employee Workload

The employee workload is a display, which lets a manager determine the department an employee is assigned to within the company and the projects the employee is currently working on. This information helps the manager assign employees to new projects that are taken on by the company. A sample employee workload screen layout is shown below:

Employee Workload

Employee Id: 11111111

Name: John Doe

Works For Department Number: 1

Name: Information Systems

Currently Assigned to:

Hours	Project Number	Project Name	Department	Weekly
	IS-101	2000 Bug	Information Systems	20
	PR-222	Inventory Control	Warehousing	5

Business rules:

1. An employee works for a single department in the company.
2. A department can have many employees working for it simultaneously.
3. A single department develops a project.
4. A department can develop many projects simultaneously.
5. An employee can be assigned to a maximum of 4 projects simultaneously.
6. A project can have many employees working on it simultaneously.
7. Employee Id, Department Number and Project Number uniquely identify employees, departments, and projects respectively.

Before applying the rules of normalization, you should have a clear understanding of the entities, relationships, and candidate keys in the view.

Entities: Employee, Department, Project

- Relationships:
1. Employee : Department
N : 1
 2. Department : Project
1 : N
 3. Employee : Project
N : M

Candidate Keys: Employee Id, Department Number, Project Number

We can now apply the normalization process to produce the schema (database design) for the view.

Refer to "Normalization Reference Sheet"

Step 1: Create the initial table

The initial table is comprised of all data items in the view. The initial table **MUST** have a primary key that uniquely identifies each row in the table. If the view contains repeating groups of attributes, they are listed in parenthesis.

The employee workload view has three candidate keys. We must select a primary key from the candidate keys. The view contains information about a specific employee; this points to the Employee Id as the primary key. We should consider all candidates though. The department number will not uniquely identify a row because a single department can have many employees assigned to it. The project number will not uniquely identify a row because many employees can work on a project. Each row in the table records information about a specific employee, the employee id will uniquely identify a row, so employee id **MUST** act as the primary key for the initial table.

A repeating group is one or more attributes that have multiple values within the view. The repeating group describes an entity that takes on the “many” role in a one-to-many or many-to-many relationship. When reviewing a source document, the repeating group will have multiple data values. A review of the screen layout shows the following attributes have multiple values and form a repeating group: Project Number, Project Name, Sponsoring Dept Name and Weekly Hours. These attributes all describe a project. Project shares a many-to-many relationship with Employee.

The initial table is shown below. A written notation is used for the initial table instead of a graphic representation of the schema (it is very difficult to depict a repeating group when using a graphic notation). The primary key attribute is underlined (EmployeeId). The repeating group of attributes appears within parenthesis.

Employee(Employee Id, Name, Department Number, Department Name, (Project Number, Project Name, Sponsoring Department Name, Weekly Hours))

Step 2: Apply the rules of 1NF

1NF has two rules:

1. A table **MUST** contain atomic attributes.
2. A table cannot contain any repeating groups of attributes.

If composite attributes exist in the view they are replaced by two or more atomic attributes. This is tempered by the business situation. In some cases, it makes sense to store the data in composite form. Some designers overlook the atomic requirement.

If a repeating group of attributes exists in the view:

- Remove the repeating group of attributes from the original table and place them in a new table.
- Duplicate the primary key of the original table and place in the new table as a foreign key relating rows between the original and new tables.
- Designate the cardinality of the relationship between the new table (child) and the original table (parent).
- Designate a primary key for the new table.

Looking at the relationship between the entities involved helps set the primary key of the new table. A one-to-many relationship between the original table's entity and the entity of the new table usually results in a single attribute primary key in the new table. A many-to-many relationship between the original table's entity and the entity of the new table usually results in a composite primary key in the new table.

Reviewing the Employee Workload view, the following violations to 1NF exist:

1. Name is a composite attribute.
2. Project Number, Project Name, Sponsoring Dept Name, and Weekly Hours form a repeating group.

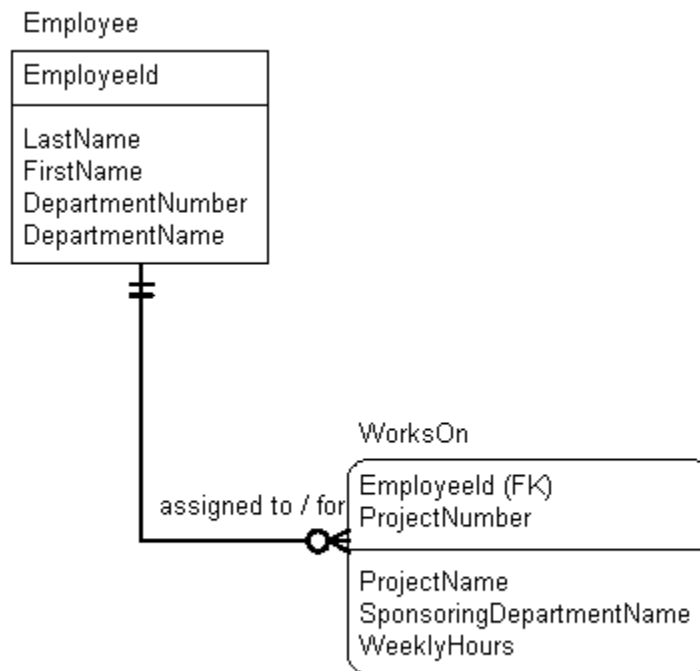
The schema is altered to comply with the rules of 1NF.

The Name attribute, in the Employee table, is replaced with LastName and FirstName.

The repeating group of attributes is removed from the Employee table and placed in a new table named WorksOn. The EmployeeId attribute (primary key from the original table) is duplicated and placed in the WorksOn table where it acts as a foreign key relating rows in the

Employee table (parent) with rows in the WorksOn table (child). Each row in the WorksOn table records information about a project that an employee is working on. The business rules state: One employee can work on a maximum of four projects. Therefore a one-to-many relationship exists between the Employee table and the WorksOn table. The cardinality of the relationship is marked as such. The last step is to designate a primary key for the WorksOn table. There is a many-to-many relationship between Employee and Project so we can expect a composite key. The candidate key attributes are considered: EmployeeId and Project Number. Since one employee can work on multiple projects several rows within the WorksOn table can have the same EmployeeId. EmployeeId cannot uniquely identify a row in the WorksOn table. Since one project can be assigned to many employees several rows within the WorksOn table can have the same Project Number. Project Number cannot uniquely identify a row in the WorksOn table. The combination of EmployeeId and Project Number will uniquely identify a row in the WorksOn table and act as the primary key for the WorksOn table.

The database design, for the employee workload query view, is shown below.



The view schema complies with the rules of first normal form; the schema is said to be in first normal form.

The design enables many employees to be assigned to a single project and many projects to be assigned to a single employee (via the composite key in the Works On table). The database design is a partial implementation of the business rule **a single employee can be assigned to a maximum of 4 projects simultaneously**. To fully implement this rule the logic of the program that adds a row to the WorksOn table will have to ensure there are a maximum of 4 rows for any given employee in the table.

Some business rules can be implemented via the DBMS in conjunction with the database design while others will need a combination of database design and program logic.

Step 3: Apply the rules of 2NF

2NF has one rule:

1. All non-key attributes in a table must **fully depend** on the entire primary key of the table.

A violation to 2NF can only occur in a table with a composite (concatenated) primary key. When evaluating a database design for violations to 2NF any table with a single attribute primary key is ignored, as a violation to 2NF cannot occur in such a table.

To evaluate a database design for compliance with 2NF an understanding of the concept of dependence is needed. What is meant by the term fully depend? How do we know if a non-key column is fully dependent on its primary key?

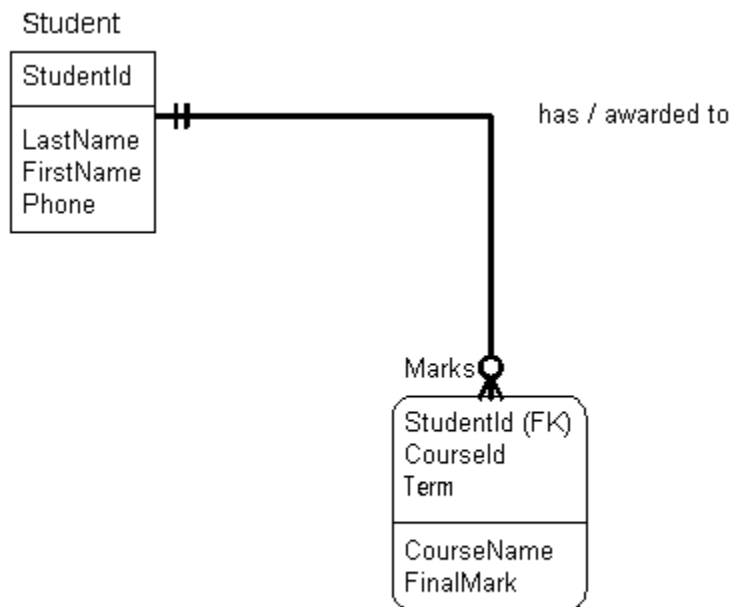
Dependence has its roots in mathematics. A detailed discussion of these principles is beyond the scope of this lesson. A pragmatic definition of dependence follows.

A non-key attribute (neither a primary key or a foreign key) is fully dependent on a primary key attribute if:

- For each value of the key attribute there can be one value for non-key attribute.

This means that if you know the value of the key attribute you know the value of the non-key attribute (or you can figure it out).

Consider the following tables:



Does this database design comply with 2NF?

Since the **Student** table has a single attribute primary key there is no possibility of a violation to 2NF. The **Marks** table has a composite primary key so this table is evaluated further. The non-key attributes are **CourseName** and **FinalMark**. Is either of these attributes fully dependent on part of the primary key? If so, it is a violation to 2NF.

Each attribute acting as part of the primary key is evaluated.

- If the value of **StudentId** (12345) is known:
 - o Is the value of **CourseName** known? No, a student can enroll in many courses so there is no way to know the value of **CourseName**. Therefore, **CourseName** is not fully dependent on **StudentId**.
 - o Is the value of **FinalMark** known? No. Therefore, **FinalMark** is not fully dependent on **StudentId**.

- If the value of CourseId (BCS245) is known:
 - o Is the value of CourseName known? Yes, if the CourseId is BCS245 the CourseName can only be one value: Database Management. Therefore, CourseName is fully dependent on CourseId and not fully dependent on the primary key of the Marks table. This is a violation to 2NF.
 - o Is the value of FinalMark known? No. Therefore, FinalMark is not fully dependent on CourseId.
- If the value of Term (2001-Fall) is known:
 - o Is the value of FinalMark known? No. Therefore, FinalMark is not fully dependent on Term.

What is FinalMark fully dependent on? A student can take a course multiple times in different terms and achieve a different mark each time the course is taken. Therefore, to know the value of FinalMark the StudentId, CourseId and Term must be known. Therefore, FinalMark is fully dependent on the primary key of the Marks table and complies with 2NF.

CourseName is fully dependent on CourseId, part of the primary key of the Marks table, and is a violation to 2NF. An attribute that violates 2NF is referred to as a **partial dependency**.

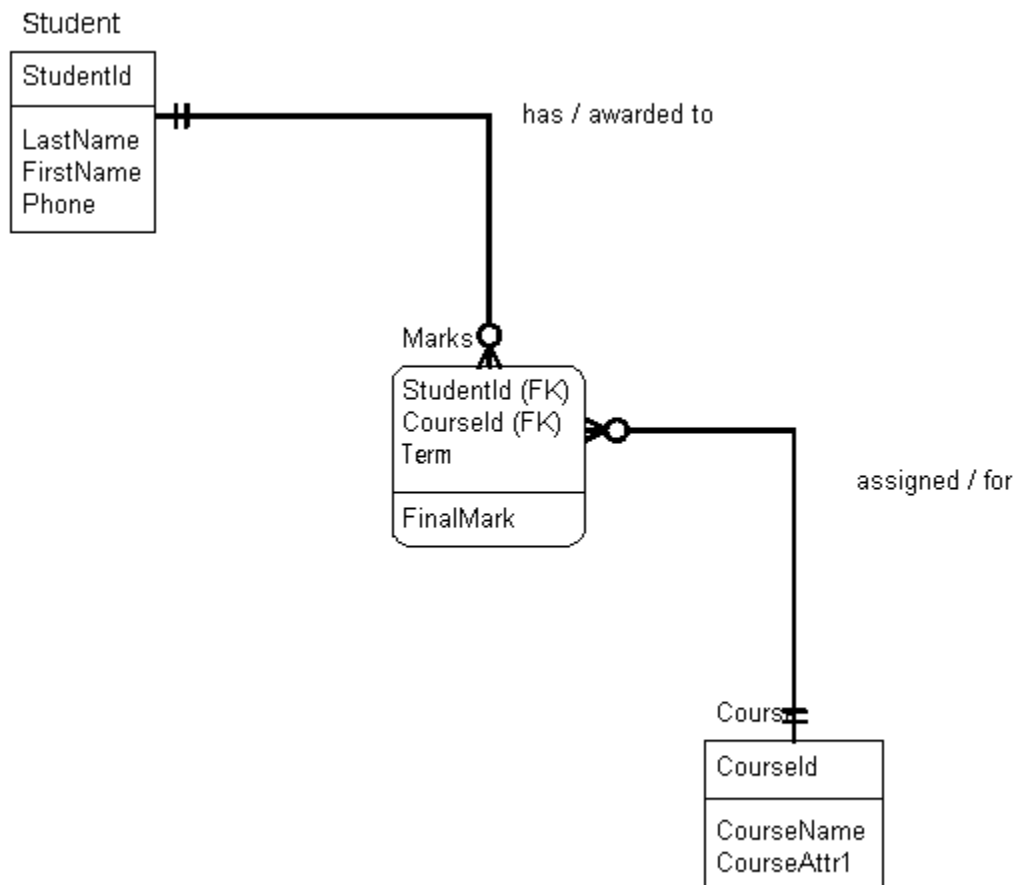
If a partial dependency exists:

- Remove the partially dependent attribute from the original table and place in a new table.
- The primary key attribute(s) the partially dependent attribute is fully dependent on, in the original table, are duplicated and placed in the new table.
- The primary key attribute(s) the partially dependent attribute is fully dependent on become foreign keys, in the original table, relating rows in the new table (parent) with rows in the original table (child).
- Designate the cardinality of the relationship between the new table (parent) and the original table (child).
- Designate a primary key for the new table.

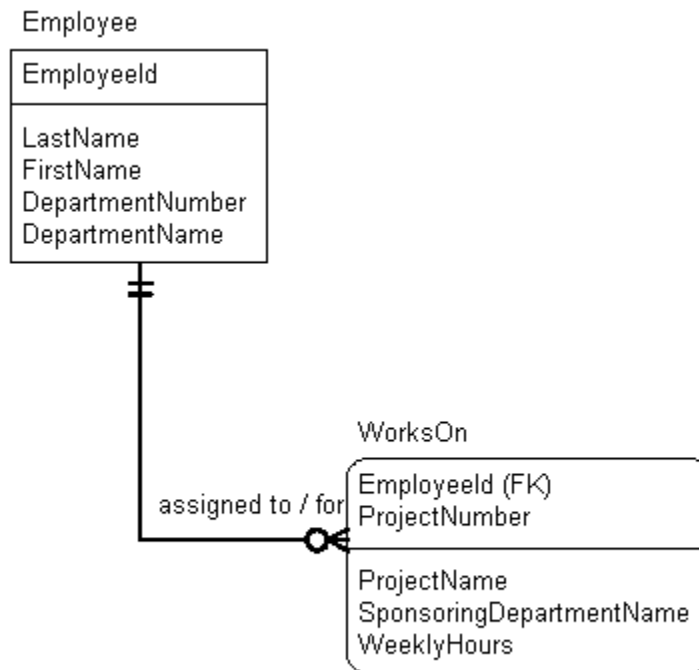
To complete the student example:

Course Name is a partial dependency (fully dependent on CourseId). CourseName is removed from the Marks table and placed in a new table named Course. CourseId is duplicated and placed in the Course table. The primary key for the Course table is CourseId.

The database design, in 2NF, is shown below.



Returning to the Employee Workload view:

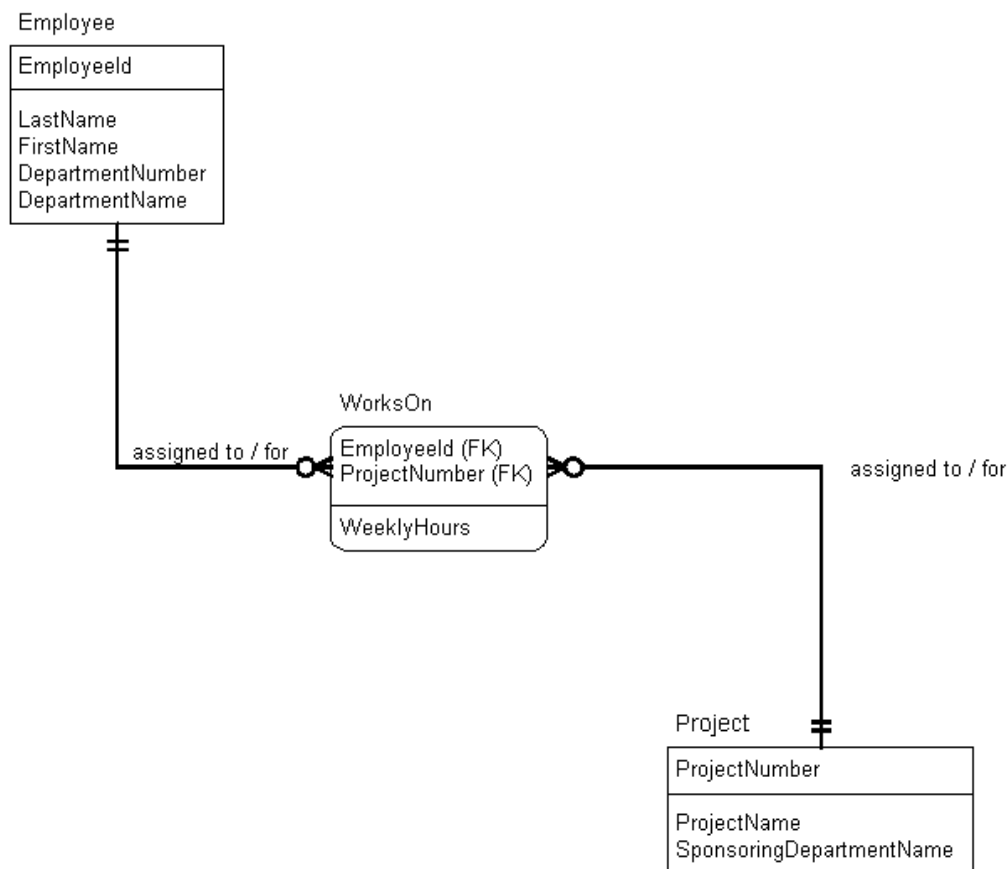


The following violations to 2NF exist:

ProjectName and SponsoringDepartmentName are both partial dependencies and are fully dependent on ProjectNumber.

Since both attributes are fully dependent on the same primary key attribute they are placed in a new table named Project. ProjectNumber is duplicated and placed in the Project table. ProjectNumber acts as a foreign key, in the WorksOn table, relating rows between the Project table (parent) and the WorksOn table (child). ProjectNumber is the primary key for the Project table.

The database design, for the employee workload query view, is shown below.



The view schema complies with the rules of second normal form; the schema is said to be in second normal form.

WorksOn is an associative entity. The many-to-many relationship between the Employee entity and the Project entity is now fully implemented as two one-to-many relationships (Employee to WorksOn and Project to WorksOn).

Step 4: Apply the rules of 3NF

3NF has one rule:

1. A non-key attribute cannot be fully dependent on another non-key attribute.

A non-key attribute that is fully dependent on another non-key attribute is termed a **transitive dependency**.

A table must have two or more non-key attributes for a violation to 3NF to occur.

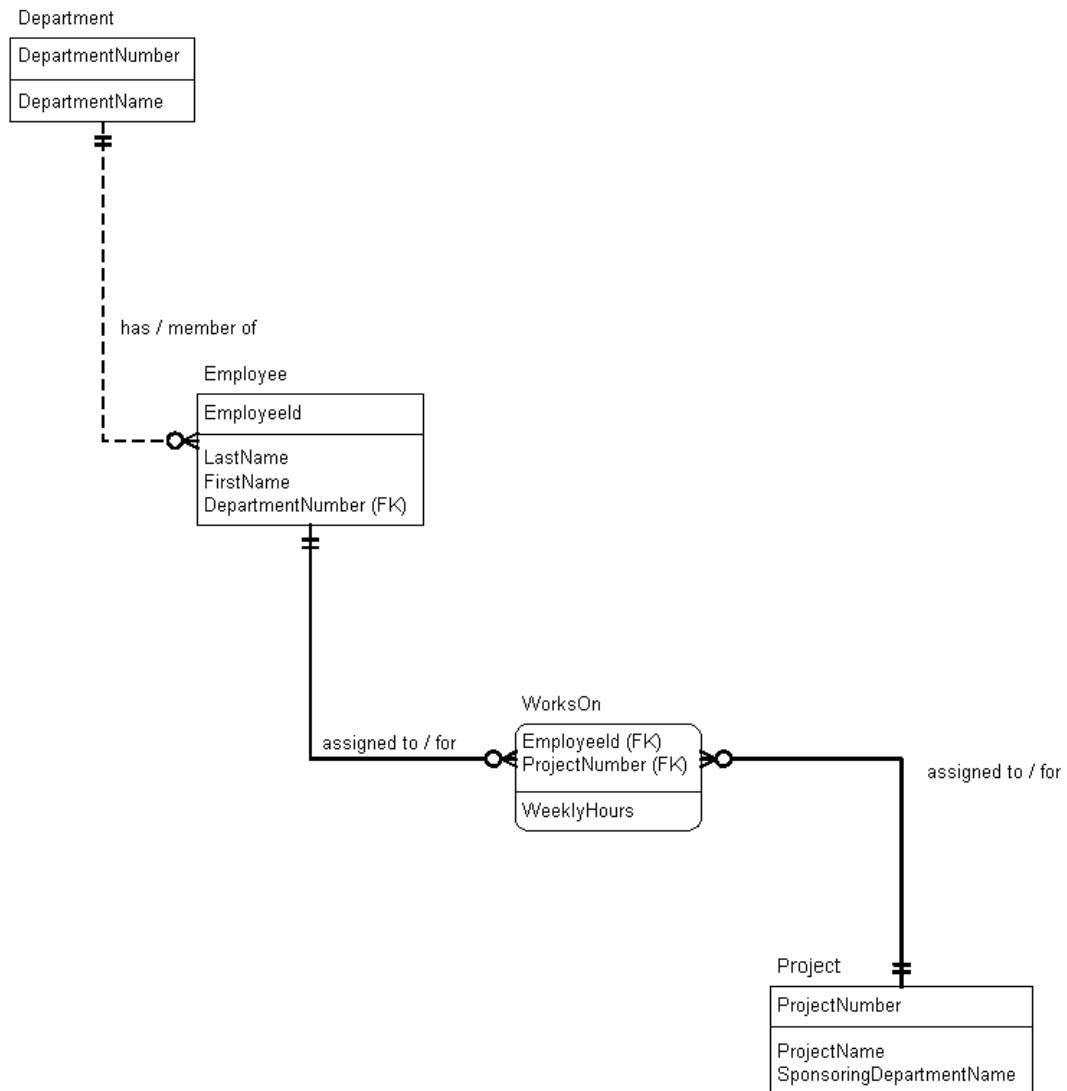
If a transitive dependency exists:

- Remove the transitively dependent attribute(s) and place in a new table.
- Duplicate the non-key attribute that the violator was fully dependent on and place in the new table.
- The non-key attribute that the violator was fully dependent on acts as a foreign key, in the original table, to relate the rows between the new table (parent) and the original table (child).
- Designate the cardinality of the relationship between the new table (parent) and the original table (child).
- Designate a primary key for the new table.

In the Employee Workload view:

Department Name is fully dependent on Department Number in the Employee table. Department Name is a transitive dependency. Department Name is removed from the Employee table and placed in a new table named Department. Department Number is duplicated and placed in the Department table. Department Number acts as a foreign key in the Employee table relating rows in the Employee table (child) to rows in the Department table (parent). Department Number is the primary key of the Department Table.

The database design, for the employee workload view, is shown below.



This completes the schema for the Employee Workload view. The schema is in 3NF. Since we do not plan to move the design further than 3NF, the logical database design for the Employee Workload view is complete. This design can be used to create the necessary tables in the database. Each entity represents one table in the database.

A similar logical database design would be created for each view/task in the application. The designs would be merged to produce a single logical database design that accommodates the needs of all tasks within the application.